



M257 Unit 12

UNDERGRADUATE COMPUTING

Putting Java to work



Sprites and sounds

Unit **12**

This publication forms part of an Open University course M257 *Putting Java to work*. Details of this and other Open University courses can be obtained from the Student Registration and Enquiry Service, The Open University, PO Box 197, Milton Keynes MK7 6BJ, United Kingdom: tel. +44 (0)870 333 4340, email general-enquiries@open.ac.uk

Alternatively, you may visit the Open University website at <http://www.open.ac.uk> where you can learn more about the wide range of courses and packs offered at all levels by The Open University.

To purchase a selection of Open University course materials visit <http://www.ouw.co.uk>, or contact Open University Worldwide, Michael Young Building, Walton Hall, Milton Keynes MK7 6AA, United Kingdom for a brochure. tel. +44 (0)1908 858785; fax +44 (0)1908 858787; email ouwenq@open.ac.uk

The Open University
Walton Hall, Milton Keynes
MK7 6AA

First published 2007.

Copyright © 2007 The Open University

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, transmitted or utilised in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without written permission from the publisher or a licence from the Copyright Licensing Agency Ltd. Details of such licences (for reprographic reproduction) may be obtained from the Copyright Licensing Agency Ltd, Saffron House, 6–10 Kirby Street, London EC1N 8TS; website <http://www.cla.co.uk/>.

Open University course materials may also be made available in electronic formats for use by students of the University. All rights, including copyright and related rights and database rights, in electronic course materials and their contents are owned by or licensed to The Open University, or otherwise used by The Open University as permitted by applicable law.

In using electronic course materials and their contents you agree that your use will be solely for the purposes of following an Open University course of study or otherwise as licensed by The Open University or its assigns.

Except as permitted above you undertake not to copy, store in any medium (including electronic storage or use in a website), distribute, transmit or retransmit, broadcast, modify or show in public such electronic materials in whole or in part without the prior written consent of The Open University or in accordance with the Copyright, Designs and Patents Act 1988.

Edited, designed and typeset for Web publication by The Open University.

WEB 868717

1.1

CONTENTS

1	Introduction	4
2	Our game	5
3	The game MIDlet	6
3.1	The other classes	6
3.2	How the game works	6
4	The Games API	7
4.1	The API classes	7
4.2	About tiled layers	7
4.3	Defining Sprites	9
4.4	The GameCanvas	11
4.5	LayerManager	14
5	Programming the game	16
5.1	Game loops	16
5.2	Our game	17
5.3	The class RobotSprite	19
5.4	How the game is started, paused and stopped	19
6	Adding sound	22
6.1	The Manager class	22
7	Further developments	24
7.1	Different screens	24
7.2	Game levels	25
7.3	Optimization	25
7.4	Portability and testing	26
7.5	Deployment	26
8	Summary	27
	Index	29

1

Introduction

In this case study we return to the world of MIDlets – software for small devices such as mobile phones. The sample application we shall explore is a very simple mobile game.

We met the core API of the Java 2 Micro Edition (J2ME) in *Unit 10*. In developing our game we use two newer extensions of J2ME:

- ▶ the Games API, which provides support for graphics including programming game backgrounds, animated objects and user interaction;
- ▶ the Media API, which supports sound.

These are part of MIDP 2.0, which supports a much richer gaming environment than MIDP 1.0.

Games programming is of extraordinary interest in itself but what the unit is intended to illustrate is the central theme of our course – that Java is recognizably the same language, with the same principles, on all platforms whether large, medium or small.

The example presented in the unit is only a demonstration prototype. At the end of the unit we will briefly survey some of the main things that would need doing to it before it could become a deliverable game for mobile phone users.

Study Note

You may wish to have the NetBeans case study project for this unit open while working through some sections.

The case study code is available as a separate download from the M257 course website and is not included on the M257 Course Software CD.

Aims

The aims of this unit are to:

- ▶ build on the knowledge of MIDP gained in *Unit 10*;
- ▶ introduce the Games API and Media API from MIDP 2.0;
- ▶ demonstrate the use of these APIs in a simple game;
- ▶ show how Java threads can be put to work in J2ME applications;
- ▶ explain the difference between blocking and non-blocking method calls;
- ▶ consider some general issues of J2ME application development.

To the user the application is what matters, so we'll start by describing it.

2 Our game

In this game the player uses the arrow keys to control a small robot (the object at top left in Figure 1) moving about in a micro-world. The goal is to reach the star portal visible near the bottom right. If the robot can successfully get to the portal before the clock runs out it will be able to escape to another dimension.

The micro-world contains obstacles – water, sandy patches and clumps of trees. The robot is unable to travel across any of these (robots are not good with such things) and can move about in the green areas only.

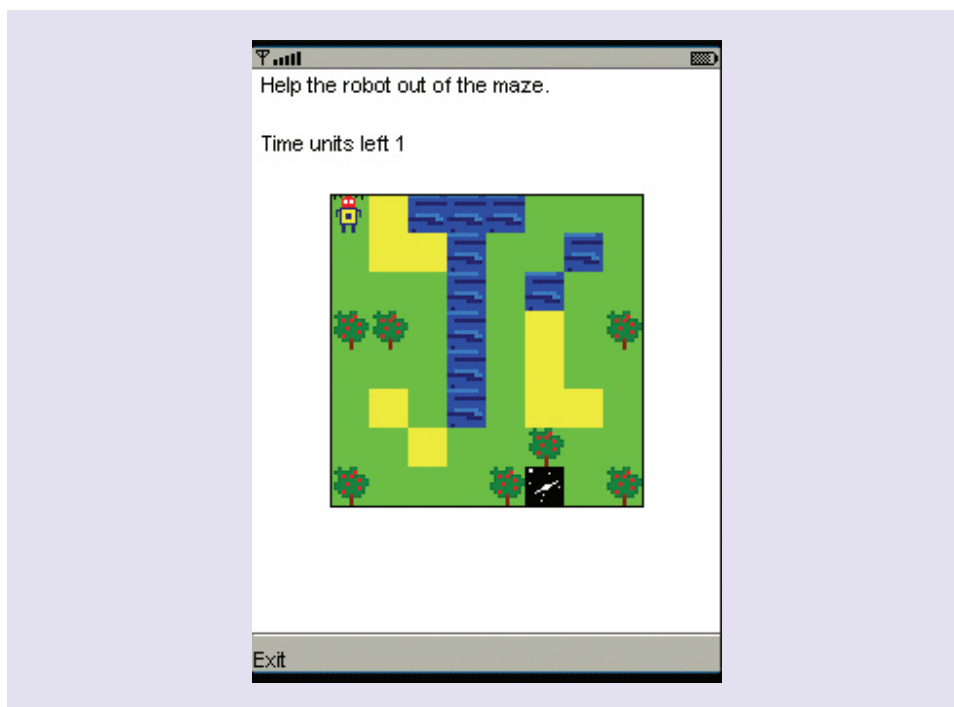


Figure 1 The micro-world of the game

3

The game MIDlet

In *Unit 10* we saw that the MIDlet life cycle has three states – paused, active and destroyed.

Remember from *Unit 10* that every mobile application written in Java is based on a MIDlet, which extends the `MIDlet` class and forms the starting point when the application is launched. When a MIDlet is first made active, its `startApp()` method is invoked. The MIDlet will create and display the various other objects that are needed for the application. It may create threads and set them running.

The MIDlet is also responsible for seeing that if the application is paused for an incoming phone call it is resumed correctly when the call ends. When a MIDlet is paused its `pauseApp()` method is invoked. When it resumes, its `startApp()` method is invoked again.

At the end, when the application finishes, the MIDlet must free any resources it has been using and stop any threads it has started. This cleaning up happens when the MIDlet's `destroyApp()` method is invoked.

In our case study application the MIDlet is called `U12MIDlet`.

3.1 The other classes

Our game has just two other classes:

- ▶ `Game`, an instance of which represents the game itself and appears as the screen shown in Figure 1. It implements `Runnable` so it can run as a separate thread.
- ▶ `RobotSprite`, an instance of which represents the robot that the player controls.

3.2 How the game works

When the application runs, the MIDlet creates an instance of `Game`, displays it and starts it. This object then creates all the other objects used in the game, acts as the container in which they are displayed, and handles all the animation and user interaction.

If the application is paused at any point, the MIDlet tells the game to suspend itself. Once the interruption has passed, the MIDlet notifies the game thread that it should resume running from the point where it was stopped.

The game screen has an exit command attached to it. The command listener for this command is the MIDlet. If the user presses the Exit key at any point, control is passed to the MIDlet's `commandAction()` method, which invokes the MIDlet's `destroyApp()` method to kill the application.

4 The Games API

The game screen is the core of the user's experience – after all, the game is why they are running our application in the first place!

To understand how the game screen is programmed we begin by outlining the Games API and how its classes are used. Then we will move on to see how games are constructed in general and how our own particular game works.

4.1 The API classes

The MIDP 2.0 Games API provides the following new classes in the package `javax.microedition.lcdui.game`:

- ▶ `TiledLayer`, which lets us define game backgrounds as a series of layers;
- ▶ `Sprite` class, which lets us program animated sprites;
- ▶ `GameCanvas`, which acts a container for the game objects, displays all the graphics and handles user input;
- ▶ `LayerManager`, which deals with the management of tiled layers and sprites.

The low level interface `Canvas` was introduced in *Unit 10*.

`TiledLayer` and `Sprite` inherit from an abstract class `Layer`.

`GameCanvas` inherits from the low-level interface `Canvas`.

The following very rough analogy may help show how these classes fit in. Think of a game canvas as a puppet theatre, which provides the stage where the action takes place and supplies all the lighting effects. Tiled layers are scenery backdrops that can be shifted and moved. Sprites are actors and props. The layer manager is a stage manager coordinating the actors and scenery.

Before we can describe our game program we need to look at how these classes are used. When you read the following sections, concentrate at first on trying to get the broad picture: what each class does and what its main methods are. Don't try to remember all the technical details for now; you can come back to them later.

4.2 About tiled layers

`TiledLayer` allows us to define large backgrounds by building them from a small set of basic tiles. Once a `TiledLayer` has been created it can be placed where we want and `TiledLayers` can also overlay one another. So in complicated games we can have multiple moving backgrounds.

Defining tiles

To create a `TiledLayer` we start with an image object, based on a Portable Network Graphics file. For example:

```
Image image = Image.createImage("/res/background.png");
```

Each `TiledLayer` is made up of cells, which are all the same size and shape, and arranged in a rectangular grid of rows and columns. The number of rows and columns, and the size of each cell measured in pixels, are specified in the arguments to the `TiledLayer` constructor. The image object is broken up into tiles of the same size as

the cells, working from left to right and top to bottom. Each tile is given an **index number**, starting at 1 (not 0!).

In our game we've used the image file shown in Figure 2. This is enlarged 5 times so you can see it better; the actual size is 120 pixels wide × 20 pixels high.

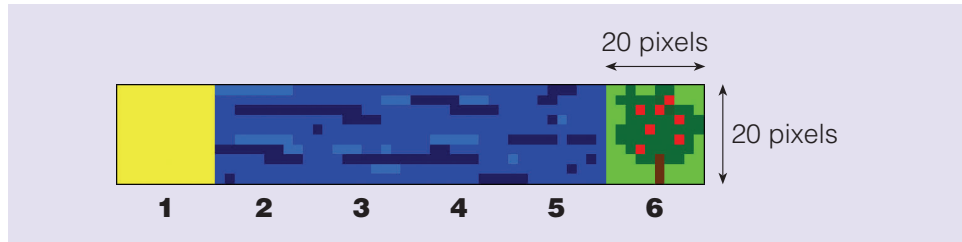


Figure 2 Tile index numbers

Constructing a TiledLayer

Note that the image width and height are both divisible by 20.

To construct a `TiledLayer` with 3 columns and 4 rows, with each cell 20 × 20 pixels, we'd use:

```
TiledLayer backgroundLayer
= new TiledLayer(3, 4, image, 20, 20);
```

This breaks the `image` up into six indexed 20 × 20 tiles: index 1 representing a patch of sand, index numbers 2–5 representing water effects (we'll explain why there are four of these in a moment) and index 6 showing a tree. The tiles are also associated with the background layer, which has 12 cells, in a grid 3 columns wide by 4 rows high.

Tiles do not have to be square, as long as they are all the same size, and the image doesn't have to be a strip, as long as it is rectangular. The tile size must fit an exact number of times into the image size, though (so we couldn't use the image above if we wanted 15 × 15 tiles, for example).

Once we have created a `TiledLayer` we specify which tile should go in each of the cells using the `setCell(int col, int row, int tileIndex)` method. For example:

```
backgroundLayer.setCell(3, 2, 6);
```

places a copy of tile index 6 (the tree) at column 3 row 2. A given tile can be used in as many cells as we like, although each cell contains only one tile.

It would be very tedious to fill large backgrounds by having a separate program statement for every cell! Instead we define an integer array, which represents the tile index to be used for each cell. If the cell is to remain empty we use 0 for the tile index. Then we loop through the array setting the cells as we go. Figure 3 shows an array and the resulting `TiledLayer`.

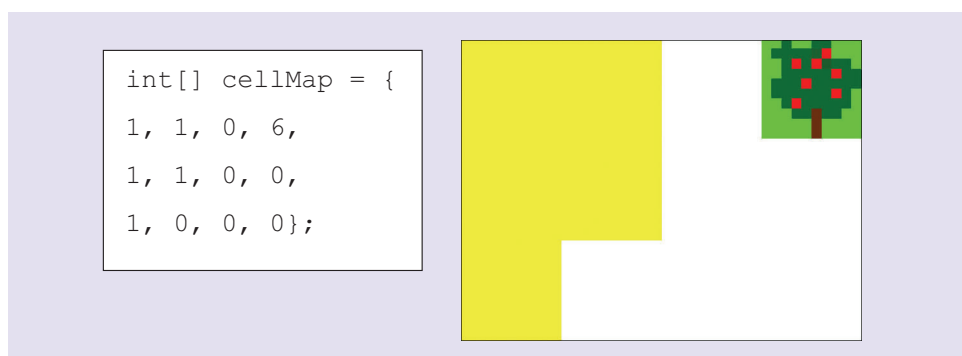


Figure 3 An array and the resulting Tiledlayer

Animated tiles

To make backgrounds a bit more interesting, tiles can be animated. Animated tiles are assigned *negative* index numbers: -1, -2 and so on. In our game the water is animated to create an impression of waves. To do this we first define an animated tile associated with the 'water' tile 2 in Figure 2:

```
backgroundLayer.createAnimatedTile(2);
```

This is the first animated tile, so it has index -1. In order to make the water change, all we need to do is assign a *different* water tile to animated tile -1:

```
backgroundLayer.setAnimatedTile(-1, 3);
```

This will make all the cells that have been given tile index -1 switch from showing water tile index 2 to water tile index 3. By repeatedly cycling through tiles 2, 3, 4, 5 and back to 2 the water is made to seem to ripple. This is the reason for our four water tiles in Figure 2.

If you examine the helper method `createBackground()` in class `Game` you will see how we have created an 8×8 `TiledLayer`, with areas of sand (index 1), animated water (index -1) and trees (index 6). This provides the background for our game.

See the *Unit 12 NetBeans case study project, Activity 12.1*.

4.3 Defining Sprites

Defining a `Sprite` is easier than defining a `TiledLayer`. We still begin with an image object though, and it is still broken up into equal-sized subunits. However, for a `Sprite` these are not tiles but **animation frames** (see Figure 4).

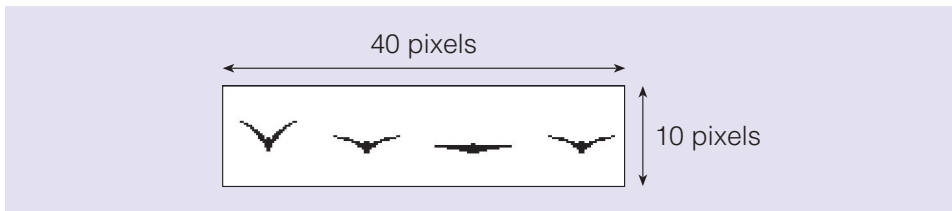


Figure 4 Animation frames

Assuming we've already used the image above to create an `Image` object referenced by `birdImage`, the constructor call:

```
Sprite birdSprite = new Sprite(birdImage, 10, 10);
```

will result in a `Sprite` object with 4 animation frames, each 10×10 pixels.

To animate the object, we repeatedly call:

```
birdSprite.nextFrame();
```

This causes the successive frames to be shown one after another like a film loop, causing the illusion of the bird flapping its wings. When the last frame is reached the animation will return automatically to the start of the sequence.

Movement

Once a `Sprite` has been constructed and placed in the game it can be moved very easily by calling `move(int dx, int dy)`. For example:

```
birdSprite.move(10, -20);
```

will move the bird 10 pixels right and 20 up from its current position.

Collision detection

In a game we often need to know if two things have collided with one another. The method `collidesWith` makes it straightforward to find out if a `Sprite` has hit something else. For example, if we wanted to respond if our bird had flown into another bird, `birdSprite2`, we would use something like:

```
if (birdSprite.collidesWith(birdSprite2, false))
{ // take appropriate action}
```

The second argument to the method indicates how we want the collision checked. Every `Sprite` is enclosed in a **collision rectangle**. By default this is just the bounding rectangle of the `Sprite`, although it can be resized and/or shifted at will using the `defineCollisonRectangle` method (Figure 5).

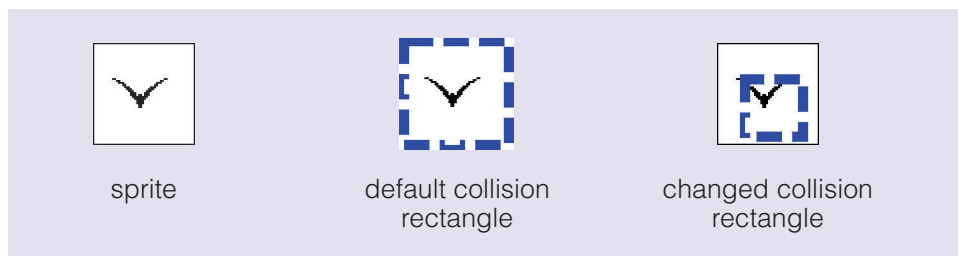


Figure 5 The collision rectangle

If the second argument to `collidesWith` is `false`, the software simply checks for a collision using the collision rectangle. If the argument is `true` a more complex scheme is used, which checks whether opaque pixels overlap; roughly speaking this means the boundary of the actual shape is used rather than the collision rectangle.

Rotations and reflections

Methods also exist for rotating sprites through a multiple of 90 degrees and for creating mirror images of them. This often lets us get away with a smaller number of animation frames than would be needed otherwise. For example, if we want to show a sprite moving forwards and backwards we just create a forward set of frames, then use the same sequence with a mirror transform to produce the backwards movement.

We don't use transforms in our game, so we won't take more space describing them here, but if you are interested you can find out more from the Games API documentation.

Creating graphics files

We are not expecting you to create any image files but you may be interested to know how it is done.

All the files we have used were created using Windows Paint and saved in Portable Network Graphics (png) format, which all MIDP implementations are required to support.

A more powerful alternative to Paint is The GIMP, available under public license. The GIMP can be downloaded from <http://sourceforge.net/> and can do far more than Paint, but is more difficult to learn to use.

4.4 The GameCanvas

Painting the display

A `GameCanvas` is where all the game objects are displayed. It has a `Graphics` object, which defines what should appear on the screen; by using this `Graphics` object we can arrange to display whatever graphics we want.

The `GameCanvas` has an **off-screen buffer** where the graphics are constructed preparatory to being displayed. The buffer is then flushed to the screen. This makes for much smoother graphics drawing.

Using `GameCanvas` graphics could not be simpler. Here's a little class (not part of the case study) that draws a red rectangle, some green text and then some blue text. For simplicity we have placed all the drawing statements in its constructor. Recall that the origin of the coordinate system for `Graphics` is the top-left corner, with axes pointing horizontally to the right and vertically down.

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class DemoCanvas extends GameCanvas
{
    private Graphics g; // lcdui.Graphics

    public GameScreen()
    {
        super(true);
        g = getGraphics();

        g.setColor(0xff0000); // red
        g.fillRect(10, 50, 50, 70); // x, y, width, height

        g.setColor(0x00ff00); // green
        String message = "The canvas width is " + getWidth();
        g.drawString(message, 10, 10, g.TOP|g.LEFT);

        g.setColor(0x0000ff); // blue
        message = "The canvas height is " + getHeight();
        g.drawString(message, 10, 30, g.TOP|g.LEFT);

        flushGraphics();
    }
}
```

In an associated `MIDlet` class we create and display an instance of `DemoCanvas`:

```
public void startApp()
{
    DemoCanvas canvas = new DemoCanvas();
    display.setCurrent(canvas);
}
```

This simple code will not work in a real game.

The result is shown in Figure 6.



Figure 6 A simple drawing on a `GameCanvas`

Each of the six digits can vary independently, apparently giving 16 777 216 colours. Mobile devices currently don't support anything like this number: 65 536 or 4096 colours are typical.

There is also a `setColor` (`int red`, `int green`, `int blue`) method, but this is less common than the single-argument version we have described.

Some explanation is needed of how the colours are set. Each colour is described by an `int`, written in the form `0x00RRGGBB`, where each of the `R`, `G` and `B` symbols stands for a hexadecimal digit in the range 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, a, b, c, d, e, f. To make the red component of the colour as large as possible, we would replace `RR` by `ff`. To have no red at all, we would replace the `RR` by `00`. The blue and green components work in the same way. So `0x00000000` switches all three components off, giving black. `0x00ffffff` turns them all up to full, giving white. Different combinations give other colours; for example `0x00ffff00` is yellow. The high-order byte of this value is ignored. For example, `0x00ff0000` is equivalent to `0xff0000`, so you may also see colours with only six hexadecimal digits.

So in our code above `0xff0000` is red, `0x00ff00` is green, `0x0000ff` is blue.

Notice that the rectangle is drawn relative to the top left of the canvas and text position is defined relative to an **anchor** `g.TOP|g.LEFT` in the same position. Text can be drawn relative to anchors in other positions, but this is the only anchor we shall be using. Notice also that there are methods that let us find the size of the canvas.

Drawing layers

To draw layers is also very simple. Adding the following code to our canvas draws a `Sprite` at position (100, 150). The `try-catch` is needed because the attempt to read the graphics file can result in an `IOException`.

```
try
{
    Image arrowImage = Image.createImage("arrow.png");
    Layer arrow = new Sprite(arrowImage);
    arrow.move(100, 150);
    arrow.paint(g);
}
catch (Exception e)
{
    e.printStackTrace();
}
```

The display will now be as shown in Figure 7.

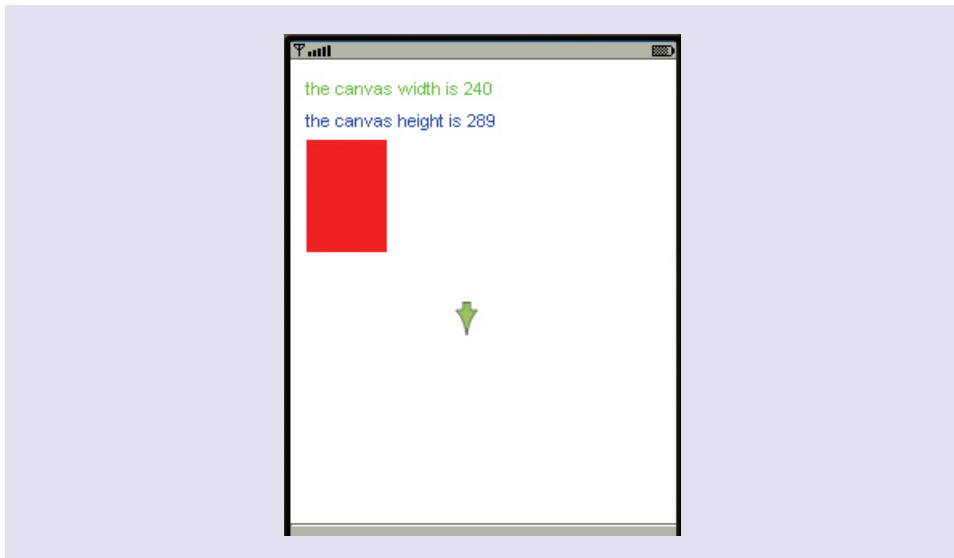


Figure 7 A **Sprite** has been added

Polling the keys

To find out whether the user has pressed any keys we use an idiom similar to the following:

```
while (true)
{
    int keyStates = this.getKeyStates();
    if ((keyStates & RIGHT_PRESSED) != 0)
    {
        System.out.println("right");
    }
}
```

This example loops round, asking the keys if they have been pressed. The information about which ones have is passed back as a number. The code `keyStates & RIGHT_PRESSED` checks to see if the right arrow key was amongst them and if so prints out a message.

Checking other keys works the same, so to check for a left arrow press we use `keyStates & LEFT_PRESSED` and so on. In our game we use only the four multidirectional keys. You can find a full list of keys from the Games API documentation.

This **polling** of the keys is convenient to use. However, to use it we have to make our game canvas run as a separate thread. To do this we modify our class as follows:

```
import javax.microedition.lcdui.*;
import javax.microedition.lcdui.game.*;

public class DemoCanvas extends GameCanvas implements Runnable
{
    private Graphics g; // lcdui.Graphics

    public DemoCanvas()
    {
        super(true);

        Thread thread = new Thread(this);
        thread.start();
    }
}
```

In *Unit 10* we described how to use the `keyPressed` method of `Canvas`.

The `getKeyStates` method in `GameCanvas` is more straightforward to use, as well as providing better responsiveness.

```

        g = getGraphics();

        // unchanged graphics drawing code goes here

        flushGraphics();
    }

    public void run()
    {
        while (true)
        {
            int keyStates = this.getKeyStates();
            if ((keyStates & RIGHT_PRESSED) != 0)
            {
                System.out.println("right");
            }
        }
    }
}

```

This uses another idiom very common in mobile applications; the **self-launching thread**. `DemoCanvas` now implements `Runnable` and has a `run()` method in which the key polling takes place. To launch itself it creates the `Thread` in which it will run, passing itself to the constructor as an argument, then starts the thread off.

4.5 LayerManager

The method of drawing layers we described above is fine if we have only a layer or two to deal with, but it would rapidly become very messy as we added more features. A `LayerManager` provides an easy way to tie all the layers together and draw them with a single statement.

For example, imagine we have variables to represent two `Sprites` and three `TiledLayers`:

```

Sprite bird1, bird2;
TiledLayer distance, foreground, middle;

```

Once we have created the objects concerned and assigned them to these variables, we create a `LayerManager` object:

```

LayerManager layerManager1 = new LayerManager();

```

Then we add the `Layer` objects to it, starting from the one we want to appear nearest to the user:

```

layerManager1.append(bird1);
layerManager1.append(bird2);
layerManager1.append(foreground);
layerManager1.append(middle);
layerManager1.append(background);

```

Now we can show them all at once!

```

layerManager1.paint(g, 0, 0);

```

When they are drawn they will overlay one another (like a stack of photographs), with the first one that was appended on the top of the stack and the last one that was appended at the bottom.

A `LayerManager` has its own coordinate system; by default the top left-hand corner is the origin (0, 0). The appended layers are actually placed on the `LayerManager`, so that their position is relative to the `LayerManager`'s origin.

If we move a `Sprite` to position (25, 40), say, it will be 25 pixels to the right and 40 pixels down from the `LayerManager`'s top left-hand corner. The `LayerManager` in turn can be painted where we like relative to the display, i.e. the top left-hand corner of the device screen.

For example, imagine the following code is executed:

```
bird1.move(25, 40);  
layerManager1.append(bird1);  
layerManager1.paint(g, 30, 50);
```

Figure 8 shows the resulting relationship between the display origin, the `LayerManager` origin and the position of the `Sprite`.

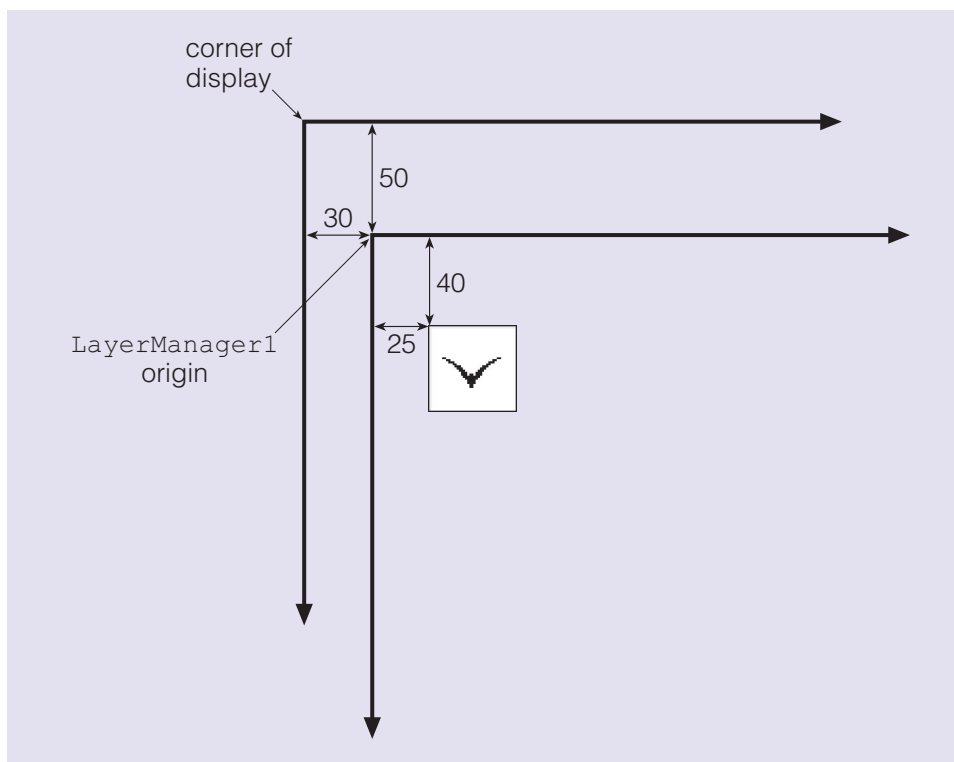


Figure 8 The coordinate system of the `LayerManager`

5

Programming the game

The previous section has outlined how the classes of the Games API are used. Now we look at the detail of how the case study game screen is programmed.

5.1 Game loops

A small stand-alone game such as ours is normally built according to the plan below. There is an initialization phase, then a **game loop** that runs until the current play of the game finishes.

Initialization:

Get a graphics object to use in painting the screen.

Create and initialize the background, sprites and any other objects.

Deploy the various objects for the start.

Game loop:

Loop until the game ends.

 Paint the game on the screen.

 Increment the game count (the number of 'ticks').

 Wait for a short time.

 Animate game objects as required.

 Check for and process collisions between game objects.

 Check for and process user input.

 Do anything else required.

End game loop.

5.2 Our game



Activity 12.1 The Robot Game

Our class `Game` follows the pattern above. The core of its initialization section is reproduced below:

```
// get a reference to the graphics
g = getGraphics();

// the game will count down from 125
gameTicks = 125;

try
{
    // get the robot and starry portal images
    // an IO exception may occur, hence the try-catch
    robotImage = Image.createImage("/res/robot.PNG");
    starPortalImage = Image.createImage("/res/portal.PNG");
}
catch (Exception e)
{
    e.printStackTrace();
}

// create the sprites
robot = new RobotSprite(robotImage, ROBOT_WIDTH, ROBOT_HEIGHT);
starPortal = new Sprite(starPortalImage);

// make the robot's collision rectangle one pixel smaller
// all round, otherwise it can't negotiate the obstacles
robot.defineCollisionRectangle
    (1, 1, robot.getWidth() - 1, robot.getHeight() - 1);

// move the star portal to the right square
starPortal.setPosition(100, 140);

// create the layer manager
layerManager = new LayerManager();

// add the sprites to the layer manager
layerManager.append(robot);
layerManager.append(starPortal);

// create the background using a helper method
background = createBackground();

// add the background to the layer manager
// it will be behind the sprites because it was appended later
layerManager.append(background);

// the first tile of the animated water
animatedWaterTileIndex = 2;
```

This uses the helper method `createBackground()` mentioned previously. The method returns the background as a `TiledLayer` of 8×8 cells, each 20×20 pixels. Placing the code that does this in the helper method helps to streamline the initialization code.

The game loop is as follows:

```
while (gameTicks >= 0) // loop until time runs out
{
    // display the graphics
    draw(g);

    // decrement the game count
    gameTicks--;

    // wait a little before the next frame otherwise
    // the animation will be too rapid
    try
    {
        Thread.sleep(FRAME_DELAY);
    }
    catch (Exception e)
    {
        e.printStackTrace();
    }

    // animate the robot
    robot.nextFrame();

    // cycle water through tiles 2, 3, 4, 5
    animatedWaterTileIndex++;
    if (animatedWaterTileIndex > 5)
    {
        animatedWaterTileIndex = 2;
    }
    background.setAnimatedTile(-1, animatedWaterTileIndex);

    // check the key states
    checkKeyStates();

    // check for collisions
    checkCollisions();

    // check robot is within bounds
    checkRobotInBounds();
}
```

This uses several helper methods:

- ▶ `draw(g)` is responsible for painting everything visible on screen. The various steps involved have been gathered together into a single method so the main game loop logic will be easier to follow.
- ▶ `checkKeyStates()` checks whether the user has pressed a key; if so, an appropriate message is sent to the robot, telling it to move and in what direction.
- ▶ `checkCollisions()` checks whether the robot has bumped into the background; if it has, its last move is undone, putting it back where it was before. This guarantees that the robot cannot move over the background.
- ▶ `checkBounds()` checks whether the robot has moved outside the boundary of the micro-world. If it is out of bounds, its last move is undone. This guarantees that the robot is confined to the micro-world.

5.3 The class RobotSprite

The reason we have had to define a class `RobotSprite` and can't simply use `Sprite` is that we want the robot to have a method `unmove()`, which if used immediately after a `move()` operation will undo the move.

This is needed so that if the program detects that the robot has been moved into forbidden territory – the background or anywhere outside the boundary of the micro-world – it can be made to instantly back off again. Since this all happens before the next screen update, the effect is that the display will never show the robot at a location where it is not allowed to be.

5.4 How the game is started, paused and stopped

Starting

Like the class `DemoCanvas` discussed in Subsection 4.4, `Game` defines a self-starting thread. The `Game` constructor creates a `Thread` instance, passing the new `Game` object itself as an argument to the `Thread` constructor, then starts this thread.

When the `MIDlet` creates a new `Game`, the latter starts itself unaided. The `MIDlet` then makes the game the current display.

Pausing

The `MIDlet` can be paused by an incoming phone call at any time and our game must obviously cater for such interruptions, otherwise users probably won't want to play it!

When an incoming call ends, the `MIDlet`'s `startApp()` method will be called again. If we have simply written:

```
public void startApp()
{
    Game game = new Game();
    display.setCurrent(game);
}
```

then it will create a completely new instance of `Game` from scratch, instead of letting the user pick up the original game again where they left off.

To overcome this we use a variable to remember whether the game has already started, and if it has we don't create a new `Game` instance.

```
private boolean started = false;
...
public void startApp()
{
    if (!isStarted()) // if the MIDlet has not run yet
    {
        game = new Game(this);
        Display.getDisplay(this).setCurrent(game);
        setStarted(true); // record game has been started
    }
    // else resume the original game
}
```

This works as we want, but there is still another problem to deal with. While the user is taking the phone call, the game thread will continue running! So when they return to the game they will discover it has moved on without them.

Obviously this is not acceptable, so to prevent it we must stop the game thread when the application is paused and restart it when the application resumes. This is done as follows. In `U12MIDlet` we include suitable statements in `startApp()` and `pauseApp()`.

```
public void startApp()
{
    if (!started) // if the MIDlet has not run yet
    {
        // start the game as shown previously
    }
    else // resume the original game
    {
        synchronized (game)
        {
            game.setRunning(true); // ask Game to restart
            game.notify();
        }
    }
}

public void pauseApp()
{
    // request that the game pause
    game.setRunning(false);
}
```

The `setRunning()` method used in `pauseApp()` modifies a boolean variable `running` in `Game`. This variable keeps track of whether the game should be running or not. Each time the game loop in `Game` is executed, the thread checks this variable and if it is set to `false`, the thread calls `wait()` to put itself in a blocked state, so pausing the game. The game loop from `Game` is shown next:

```
while (!isRunning())
{
    synchronized (this)
    {
        try
        {
            this.wait();
        }

        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

Thus, when the MIDlet invokes `game.setRunning(false)`, the game thread calls `wait()` and blocks. It remains blocked until the MIDlet invokes `game.setRunning(true)` and then `game.notify()` to wake it up again from its `startApp()` method.

There is one new Java idiom here that you haven't met before. If `notify()` is used to wake up the thread blocked by the `wait()`, both must refer to the same lock. In *Unit 8* we used synchronized methods and the lock was the one belonging to the object the methods were invoked on. In this case we are using what's called a **synchronized block**, which has the structure:

```
synchronized (someObject)
{
    //code
}
```

Stopping

Our game will stop naturally when the number of ticks expires and the game loop finishes. But what if the user exits before that? The `destroyApp()` method of the MIDlet should be able to stop the thread at that point.

This is not the same as putting the thread into a blocked state; we want it to actually terminate. How can this be done? Well, just as we had a variable that indicated the thread should block, we can have another that indicates it should terminate. Then at the start of the game loop we include a statement that checks this variable. The `break` statement is a Java facility that allows us to immediately exit a loop.

```
while (gameTicks >= 0) // loop until time runs out
{
    if (requestedToStop)
    {
        break; // jump out of loop
    }

    // rest of game loop code
}
```

If `requestedToStop` is true then the game loop will abruptly terminate. Of course, using `break` is not the way we usually exit from loops, but in this case it makes for a simple way to end the program.

If the user presses the Exit key, the `commandAction()` method in the MIDlet calls the MIDlet's `destroyApp()` method and the latter requests the game thread to stop.

6

Adding sound

Considerate applications allow background music to be switched off!

In this section we look very briefly at the class `javax.microedition.media.Manager`, which provides facilities for adding musical notes to a J2ME application. These can be built up into simple tunes, which can be played in the background while users are browsing menus or playing games, or simply provide a signature tune when the application is launched. We can also make up short tunes to play in response to particular game events.

6.1 The Manager class

Playing a note

The `Manager` class has only static methods. The one we are interested in here is `playTone(int note, int duration, int volume)`. Its parameters are as follows:

- ▶ `note` is the musical pitch in the range 0–127;
- ▶ `duration` is the length in milliseconds;
- ▶ `volume` is the loudness on a scale of 0–100.

The pitch number uses standard Musical Instrument Digital Interface (MIDI) values, in which 69 represents A and each step is a semitone.

Using this couldn't be simpler. All we need to do is import `javax.microedition.media.Manager`, then call the method whenever we want to play a note.

For example:

```
Manager.playTone(72, 125, 50);
```

will play note 72 for 125 milliseconds at half volume.

Playing a tune

Playing a single note is quite straightforward but how do we play a tune, which is a sequence of notes? At first sight all we have to do is use several `playTone` calls, one after another, like this for instance:

```
Manager.playTone(60, 125, 50); // line 1
Manager.playTone(70, 125, 50); // line 2
Manager.playTone(80, 125, 50); // line 3
```

Unfortunately this doesn't work because `playTone` is a **non-blocking call**.

Usually when a thread calls a method it can't do anything else while the method is executing. Only when the method finishes can the thread execute other code. This is a **blocking call**.

With a non-blocking call the thread *isn't* delayed and can go on immediately to execute other code. Meanwhile the method executes in parallel.

So when line 1 above is executed the first note starts playing; however, the program does not wait until the 125 milliseconds are up but proceeds at once to line 2. This sets the second note sounding, and then the program executes line 3 still without any delay. Now all three notes are sounding at once – not at all what we wanted!

To play the three notes separately as intended, we need to make the thread wait until each note has finished before playing the next, like this:

```
Manager.playTone(60, noteLength1, 50); // line 1
Thread.sleep(noteLength1);
Manager.playTone(70, noteLength2, 50); // line 2
...
```

You can see an example in the static method `endTune` in the case study class `Game`. The method plays a three-note tune several times in succession. The idea is that this jingle gets played if the tick count reaches zero and the user runs out of time!

Background music

The technique we have outlined above works well when we want to play notes in response to things that happen in a game. But what if we want background music that will play independently? In that case we define a runnable class to take care of the music, and launch it in a separate thread. This music thread will need methods for pausing and stopping it, like those we discussed in Subsection 5.4 for the game thread.

Non-musical sounds

If we want non-musical sounds we have to use audio clips. Here's the code that produces a suitable sound effect if the robot reaches the portal and is whisked off to another dimension:

```
InputStream is = getClass().getResourceAsStream("/res/whoosh.
wav");
Player player = Manager.createPlayer(is, "audio/X-wav");
player.start();
```

The first step is to create an input stream for reading the audio file. The stream is then passed to a static method in class `Manager`, with a second argument specifying what media format is used, in this case a wav file. The method returns an instance of class `Player`. Invoking `start()` on this object plays the audio clip.

Unfortunately not all devices support audio playback.



Activity 12.2

Modifying the Robot Game

We saw how to create input streams in *Unit 10*.

7 Further developments

The game we've developed is a simple prototype – a working model that takes an idea and shows that it could be implemented. An enormous amount of development would be needed before we had something that could be a commercial product, or even offered as freeware. In this section we outline some of the issues that would need addressing. Most of these would be relevant to any application for small devices, not just games.

7.1 Different screens

A mobile application normally consists of a number of linked screens, which users can move between. At present our undeveloped game has just one screen. Figure 9 below shows what we'd expect to find in the finished product.

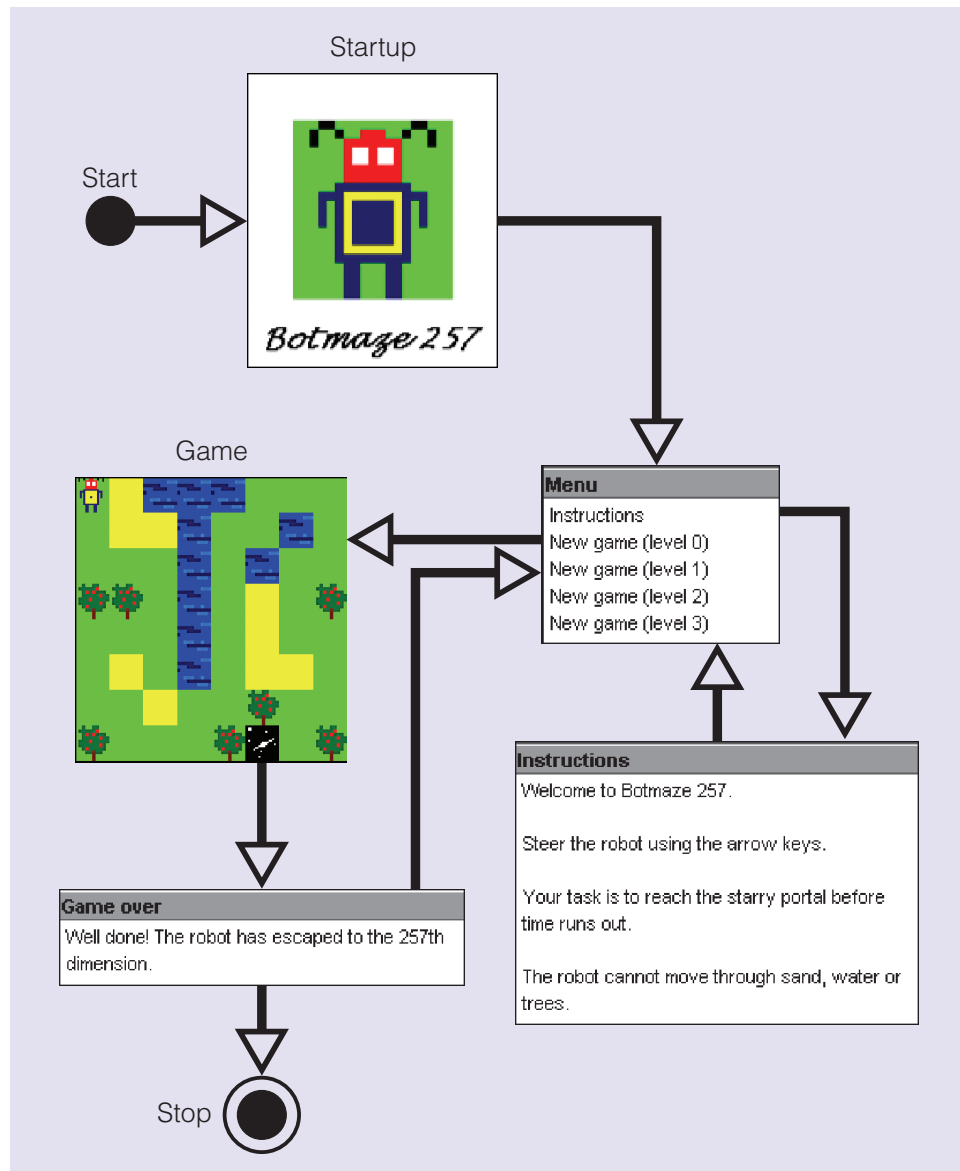


Figure 9 Linked screens in a game

One way to coordinate screen transitions is for the MIDlet to pass a reference to itself to the other objects it creates. The other screens can then use this reference to call back the MIDlet when necessary. For example, when a game ends the game screen can invoke a suitable method on the MIDlet, and the latter will then create and display a 'game over' screen.

7.2 Game levels

Anyone playing our simple game is likely to find it interesting for a short time, because it presents a modest challenge. However, once the user has played it a few times and become reasonably skilful their interest will quickly fade. We will need to program a series of graduated game levels they can progress through. The levels should get steadily harder, each one offering extra challenges and adding variety to the micro-world, so the user's interest continues to be high.

In a game like ours, what will usually vary are things such as the layout of the playing areas and the location of the start and finish points. These details will normally be stored in the long-term memory of the mobile device, and since this persistent memory is very limited we need to think about how to conserve it.

The alternative is to download the data on demand.

A common technique for storing game levels is to represent a screen as a series of bytes. For example, the initial game screen we have been working with might be stored as shown in Figure 10.

r	s	w	w	w	b	b	b
b	s	s	w	b	b	w	b
b	b	b	w	b	w	b	b
t	t	b	w	b	s	b	t
b	b	b	w	b	s	b	b
b	s	b	w	b	s	s	b
b	b	s	b	b	t	b	b
t	b	b	b	t	p	b	t

Figure 10 A games screen represented as a series of bytes. Key: **r** = robot, **s** = sand, **w** = water, **t** = tree, **p** = portal, **b** = background

This can be stored in a file. When the corresponding screen is needed, the file is read into memory then processed to produce the required layout.

7.3 Optimization

A mobile device has very limited capability and resources are always very tight, so once our application is written we may want to look at ways it could be optimized.

There are two aspects to optimization. Firstly, we want to keep memory heap size low. Secondly, if the game shows signs of becoming unresponsive it may be necessary to consider ways of reducing the amount of processing required.

There are tools available for monitoring resource usage, so it's possible to measure the effect of changing the program in various ways. However, we must always remember that mobile platforms vary widely, so we can't guarantee that an apparent improvement in performance will apply across all platforms.

However, we can state a few general guidelines for writing efficient mobile code.

To keep memory use down:

- ▶ Don't create objects unnecessarily, and as soon as an object is no longer needed destroy it by setting all references to it to `null`.
- ▶ Use `StringBuffer` objects rather than strings.

To reduce processing load:

- ▶ Use local variables instead of instance variables whenever possible; local variables are faster to access.
- ▶ Use `static final` methods where possible.
- ▶ Make sure loops don't include statements that could have been placed outside them.

Methods which are declared `static` or `final` will run faster because the virtual machine does not have to work out at runtime what code to execute, since `static` or `final` methods can't be overridden in subclasses.

A similar issue is **localization**. If we want the game to be used internationally we will need to offer different versions of the program for use in different countries.

7.4 Portability and testing

Our game has been written to run on a device that supports colour, has some sound capability and has a display of at least 160×160 pixels. But many devices will be much more limited than this. What should happen then?

An application can find out the details of the platform it's running on and adapt accordingly. For example, if the dimensions of the screen were smaller than 160×160 our game could decide to use a different layout. Mobile applications will normally do their best to work across a variety of platforms.

However, if the platform is too different from what was expected an application may simply not be able to run sensibly. In this case it should throw an exception. It's most important that this is caught and handled appropriately; users won't appreciate an application just stopping unexpectedly. The program should display an alert, saying what the problem is and suggesting a possible solution: for example, offering a URI from which an alternative version of the software can be downloaded.

The huge diversity of mobile platforms is also an issue for testing. An application developed using an emulator is not automatically guaranteed to function correctly on other platforms. If we were developing a real game we would need to try it out on a range of target devices.

7.5 Deployment

The most likely way for users to install our game is via WAP. The application would first need to be packaged in a `.jar` and a `.jad` file as described in Subsection 3.6 of *Unit 10*. These files would be placed on an internet server, along with a web page written in Wireless Markup Language (WML). Users would then direct their phones to the web page and download the game.

8

Summary

In this case study we looked at a prototype of a game for mobile phones. We introduced some of the new classes in the Games and Media APIs from MIDP 2.0, and showed how they would be used in a typical animated game. We also looked at multithreading and how threads can be started and stopped when required. The concept of blocking and non-blocking calls was briefly discussed. Finally we touched on a range of issues that would need to be resolved before a prototype could be brought to a deliverable product.

LEARNING OUTCOMES

When you have completed this unit, you should be able to:

- ▶ understand the basic principles of animation;
- ▶ use the classes from the MIDP 2.0 Games API in simple applications;
- ▶ write self-starting thread classes, and provide methods for pausing and stopping threads;
- ▶ use the MIDP 2.0 Media API to play simple tones and audio clips;
- ▶ understand the difference between a blocking and a non-blocking method call;
- ▶ appreciate some of the issues involved in developing deliverable products for small devices.

Concepts

The following concepts have been introduced in this unit:

anchor, animation frame, blocking call, collision detection, collision rectangle, `GameCanvas`, game loop, index number, `LayerManager`, localization, non-blocking call, off-screen buffer, polling, self-launching thread, `Sprite`, synchronized block, `TiledLayer`.

Index

A

anchor 12

animation frame 9–10

API 4

B

blocking call 22

C

collision detection 10

collision rectangle 10

G

game loop 16

GameCanvas 7

I

index number 8

J

J2ME 4

Java 2 Micro Edition 4

L

LayerManager 7

M

MIDlets 4

N

non-blocking call 22

O

off-screen buffer 11

P

polling 13

S

self-launching thread 14

Sprite 7, 9–10, 12–15, 17, 19, 28

synchronized block 21

T

TiledLayer 7